

# 目錄

|                |         |
|----------------|---------|
| 介绍             | 1.1     |
| GitBook        | 1.2     |
| 代码组织规范         | 1.3     |
| 项目结构           | 1.3.1   |
| Vue 文件编写代码规范   | 1.3.2   |
| 项目注释规范         | 1.3.3   |
| Git 提交规范       | 1.3.4   |
| 数据中心           | 1.3.5   |
| State          | 1.3.5.1 |
| Mutations      | 1.3.5.2 |
| Mutation.types | 1.3.5.3 |
| Actions        | 1.3.5.4 |
| Getters        | 1.3.5.5 |
| Index          | 1.3.5.6 |
| 路由规范           | 1.3.6   |
| 生产代码讲解         | 1.4     |
| 项目前期准备         | 1.4.1   |
| 首页             | 1.4.2   |

# Vue - 项目开发规范

一份关于 Vue 项目多人开发协作遵守的规范。

记住：这是一份文档，不是一个 npm 的安装包。很多时候你不需要安装下载它，你需要的只是最终的团队规范，所以只要下载 ./dist 目录下的文件即可。

## 安装

如果你对此项目感兴趣，并且希望它变得更好，更完善，可以尝试克隆到本地，进行开发调试，非常欢迎

## 下载

```
git clone https://github.com/184455/vue-team-develop-document.git
```

## 安装依赖

```
cd vue-team-develop-document  
gitbook install
```

## 运行

```
gitbook serve
```

打开浏览器访问：<http://localhost:4000/>

我们默认你已经安装了：node.js, gitbook 等工具。然后你就能看到项目，并且本地调试是热更新的。

最后，如果你对 GitBook 的使用不是很熟悉，请先看：[GitBook 的使用](#)

# 这里是 GitBook 使用说明

这篇文章的内容，主要参考了这位网友的博客，感觉写的很详细，想阅读原文的可以进去看看：  
[GitBook 从懵逼到入门](#)

## 什么是 GitBook？

GitBook 是一个基于 Node.js 的命令行工具，支持 Markdown 和 AsciiDoc 两种语法格式，可以输出 HTML、PDF、eBook 等格式的电子书。所以我更喜欢把 GitBook 定义为文档格式转换工具。

## 如何使用？

使用非常简单：

1、安装 node.js；

2、安装 GitBook： `npm install -g gitbook-cli`

MAC OS 用户：`sudo npm install -g gitbook-cli`，然后输入密码，回车

3、初始化一个 GitBook 的项目：`gitbook init`

然后目录下面就会多出来两个目录：

`README.md` —— 书籍的介绍写在这个文件里

`SUMMARY.md` —— 书籍的目录结构在这里配置

有了这些文件夹，你就可以在对应目录创建自己的 markdown 文件了，开始自己的写作之路！

4、执行：`gitbook serve` 命令，在本地开启一个开发预览模式。开发体验和 Vue 的项目是一样的，编辑器编写，然后自动实现开发热加载，在浏览器中实时查看效果。

关于编辑器，只要能写就行。不过推荐使用 Markdown 的编辑器。首先语法层面支持良好，还有各种提示。还有就是和用 word 写文档一样，所见即得。

5、创作完毕，构建最终产品：`gitbook build`

`gitbook build`: 构建的是可以部署在服务器上的 HTML 文件；

`gitbook pdf`: 最终输出一个 PDF 的产品

`gitbook mobi`: 构建成电子书

后面的两种方式需要一个安装一个第三方的插件：`calibre`，自己百度安装。然后配置对应的环境路径。

## 使用过程中遇到的问题

注意：以下截图出自：[https://blog.csdn.net/Hulu\\_IT/article/details/84977390](https://blog.csdn.net/Hulu_IT/article/details/84977390)

需要复制命令的，自行点开连接。并且针对 MAC OS 的用户使用，Windows 配置环境变量，自己百度。

### 常见问题二：

- 问题描述：执行gitbook pdf ./ ~/Desktop/test/test/时，报错：InstallRequiredError: "ebook-convert" is not installed.  
Install it from Calibre: <https://calibre-ebook.com>
- 解决办法：
  - 缺少ebook-convert,需要安装操作系统对应的calibre  
下载地址：[https://calibre-ebook.com/download\\_osx](https://calibre-ebook.com/download_osx)
  - 安装该应用完成后，建立命令行链接  
`sudo ln -s /Applications/calibre.app/Contents/MacOS/ebook-convert /usr/local/bin`
  - 再次执行gitbook pdf ./ ~/Desktop/test/test/xx.pdf即可

### 常见问题三：

- 问题描述：执行命令gitbook pdf ./ ~/Desktop/test/test/时，报错：Error: EISDIR: illegal operation on a directory, open '/Users/python/Desktop/test/test/'
- 解决办法：
  - 在执行命令gitbook pdf ./ ~/Desktop/test/test/时，加上要生成的pdf文件的名字即可解决

还有一个比较关心的问题是：在项目中，我们都有一个：**package.json** 文件来管理我们的依赖，在 gitbook 中没有这个文件，取而代之的是：**book.json**。关于里面详细的配置项，不多，很简单，直接百度：**gitbook 的 book.json 配置**，出来一大堆。你需要的各种依赖，都可以在网上查找，都有现成的东西，只要做简单的配置就好。

最后放上两个自己平时用的最多的，关于gitbook 的使用，配置的连接：

- 1、gitbook 的详细介绍
- 2、gitbook 实用配置及插件介绍

# 项目代码构建规范

这主要放置代码编写时候的规范

- 1、我们会默认使用 Eslint 静态检查工具检查你的代码，该规范建立在 Eslint 的前提。
- 2、凡事涉及组件的情况，一律按照类(class)的概念处理，命名一律：大写字母开头的驼峰法。
- 3、小驼峰法：以小写字母开头的驼峰法命名。
- 4、大驼峰法：以大写字母开头的驼峰法命名。

# 目录构建规范

|                     |                              |
|---------------------|------------------------------|
| — project-name      |                              |
| — public            | 项目公共目录                       |
| — favicon.ico       | Favourite 图标                 |
| — index.html        | 页面入口 HTML 模板                 |
| — src               | 项目源码目录                       |
| — main.js           | 入口js文件                       |
| — App.vue           | 根组件                          |
| — api               | 把所有请求数据的接口，按照模块，写在单独的 JS 文件中 |
| — home.js           | 首页接口                         |
| — detail.js         | 详情页接口，等等                     |
| — ...               |                              |
| — assets            | 静态资源目录，公共的静态资源，图片，字体等        |
| — fonts             | 字体资源                         |
| — images            | 图片资源，该文件夹下还可以按照功能或模块做更细的划分   |
| — ...               |                              |
| — common            | 公共脚本，样式，配置，等动态信息             |
| — js                | 公共脚本                         |
| — utilis.js         | 公共 JS 工具函数                   |
| — config.js         | 公共配置                         |
| — ...               |                              |
| — style             | 公共样式，可以是各种预处理语言              |
| — index.css         | 样式主文件                        |
| — reset.css         | 重置样式                         |
| — ...               |                              |
| — ...               |                              |
| — components        | 公共组件目录                       |
| — confirm           | 弹框组件                         |
| — scroll            | 局部内容滚动组件                     |
| — ...               |                              |
| — http              | 封装的 HTTP 请求方法                |
| — axios.js          | Axios 的封装                    |
| — jsonp.js          | JSONP 的封装                    |
| — ...               |                              |
| — store             | 数据中心                         |
| — state.js          | state 数据源，数据定义               |
| — mutations.js      | 同步修改 state 数据的方法定义           |
| — mutation-types.js | 定义 Mutations 的类型             |
| — actions.js        | 异步修改 state 数据的方法定义           |
| — getters.js        | 获取数据的方法定义                    |
| — index.js          | 数据中心入口文件                     |
| — routes            | 前端路由                         |
| — index.js          |                              |
| — views             | 页面目录，所有业务逻辑的页面都应该放这里         |
| — home              | 应用首页                         |
| — ...               |                              |
| — ...               |                              |
| — .env.development  | Vue 开发环境的配置                  |
| — .env.production   | Vue 生成环境的配置                  |
| — .eslintrc.js      | Eslint 配置文件                  |
| — .gitignore        | Git 忽略文件                     |
| — package.json      | 包依赖文件                        |
| — package-lock.json | 依赖包版本管理文件                    |

|                   |                 |
|-------------------|-----------------|
| ├── README.md     | 项目介绍            |
| ├── vue.config.js | vue/cli 项目脚手架配置 |
| ...               |                 |

## src 文件说明

- 1、src/api 模块的请求方法，应该参考 src/views 中的直接子文件夹的结构，做映射。
- 2、src/assets 项目的静态资源文件。虽然是静态文件，比如图片，字体，等资源，但是还是通过 webpack 处理一下会好一些，因为有些比较小的文件会被打包到文件，减少请求或者压缩第三方包等。这个模块的文件，如果还需要扩展，一个单词作为文件夹名字！保持简洁性。
- 3、src/common 主要是放置以下公共的动态的脚本，样式的一个目录。这里主要放置样式和 JS。在实际中，样式可能是各种预处理语言写的，请自行组织目录结构。
- 4、src/components 公共组件。这里放置你在项目中抽象的基础，业务组件。你应该为每个组件都单独建一个文件夹，以做好组件之间的隔离，并且有一个默认的文件：index.vue 文件，以便引入组件时少写几个字。默认这里的文件是一个最小的单位，不应该有依赖其他组件，或操作 state 状态等行为。
- 5、src/http 主要是关于请求方法的封装，以封装好，建议开发过程中不要修改，因为会影响到全局。
- 6、src/store 数据中心，这部分内容比较多，独立出来，详情参考：[数据中心](#)
- 7、src/router 页面路由。默认所有路由映射写在一个文件，如果需要路由模块化，请自行组织。
- 8、src/views 所有业务逻辑的页面。这个文件下肯定有很多组件。

## 文件命名规范

- 1、所有文件和文件夹，统一使用：小写字母开头，英文中划线分隔的方式命名。**必须遵守！**
- 2、组件名应该以高级别的(通常是一般化描述的)单词开头，以描述性的修饰词结尾。[参考这里](#)

目录设计是按照功能划分成一个文件夹，以便于满足日后功能扩展的需要。同时也保证目录的简洁，可读性。

注意：这个是 vue/cli 3.0 以上脚手架的目录结构，与 2.0 之前的可能会有一点点差别，但是主要的 src 开发目录是差不多的，可以用作参考。

# Vue 文件书写规范

在我们的开发中，与我们接触最多就是：xxx.vue 的文件，所以我们应该制定一些规则来规范我们的代码。

按照一般的习惯组织代码，从上到下，依次是：template, script 和 style 标签。然后依次说明它们在编程时应该注意的问题。

## template

在模版文件中，你应该遵守 HTML 的书写规范

**1、标签语义化。**切忌清一色的 div 元素，列表可以使用 ul li，文字使用 p 标签，标题使用 h\* 标签等等。虽然 html5 推出了语义化的标签，但是它们还有很多兼容性问题，在不支持的浏览器导致布局错乱，所以，不建议在生产环境中使用：section, aside, header, footer, article，等 HTML5 布局标签。

**2、样式 class 的命名：**可以由小写字母中划线和数字组成，但必须以小写字母开头。不建议使用驼峰法命名 class 的属性。以下是一些常用到的 class 的名字。

包裹层: .xx-wrap; 列表: .xx-list; 列表项: .xx-list-item; 左边内容: .xx-left; 中间内容: .xx-middle; 右边内容: .xx-right; 某个页面: .xx-page;

**3、自定义标签：**使用自闭标签的写法：`<my-private-components/>`。[参考这里](#)

**4、多特性，分行写。**示例代码如下，具体原因 [看这里](#)

```
<scroll
  ref="scrollWrap"
  :data="homeData"
  :pullDownRefresh="true"
  :pullUpLoad="true"
  class="home-scroll-warp"
  @pullingDown="pullingDownGetNewData"
  @pullingUp="pullingUpGetMore"
/>
```

**5、模版中使用表达式，复杂情况使用计算属性或函数。**[参考这里](#)

## script

在 script 标签中，你应该遵守 Js 的规范

**1、在 script 标签的最顶部，应该是引入文件的定义。**

2、vue/cli 脚手架自带的 '@' 问题。你可以使用绝对或相对路径引入你自定义的文件，也可以使用脚手架自带的指向 src 开发目录的 '@' 符号。前者可能在写法上可能不是很美观，有一长串的地址，而且日后文件的关系变动，必须手动维护地址；后者牺牲了可读性，获得了更简洁的绝对地址。日后文件的关系变动，只要文件名不变，则不需维护路径，维护成本要小很多。但是使用 '@' 符号对编辑器不友好，点击方法，并不会直接跳转到方法定义，带来了调试和阅读的成本。权衡两者，各有利弊，使用哪种方式，可以根据个人喜好。但是有一点：**不能混用两种方法。**

3、引入组件，命名使用：首字母大写的驼峰法命名。推荐使用 ES6 的方式引入。如果它们有依赖关系，最好体现在命名上，比如：

```
import Article from 'xxx'
import ArticleDetail from 'xxx'
```

4、组件的选项，根据官方的推荐按照以下定义。[原文出处](#)

```
<script>
  export default {
    name: 'ExampleName', // 这个名字推荐：大写字母开头驼峰法命名。
    props: {},           // Props 定义。
    components: {},     // 组件定义。
    directives: {},    // 指令定义。
    mixins: [],         // 混入 Mixin 定义。
    data () {           // Data 定义。
      return {
        dataProps: '' // Data 属性的每一个变量都需要在后面写注释说明用途，就像这样
      }
    },
    computed: {},       // 计算属性定义。
    created () {},     // 生命钩子函数，按照他们调用的顺序。
    mounted () {},     // 挂载到元素。
    activated () {},   // 使用 keep-alive 包裹的组件激活触发的钩子函数。
    deactivated () {}, // 使用 keep-alive 包裹的组件离开时触发的钩子函数。
    watch: {},          // 属性变化监听器。
    methods: {          // 组件方法定义。
      publicFunction () {} // 公共方法的定义，可以提供外面使用
      _privateFunction () {} // 私有方法，下划线定义，仅供组件内使用。多单词，注意与系统名字冲突!
    }
  }
</script>
```

5、Data 必须是一个函数。[参考这里](#)

6、Props 定义细则。[参考这里](#)

7、为 v-for 设置 key 值。[参考这里](#)

8、避免 v-if 和 v-for 用在一起。[参考这里](#)

## style

样式部分，因为样式有原生的 CSS 写法，也有使用预处理语言：Sass, Stylus, Less, PostCSS，所以情况比较多，也比较复杂。在我实际的工作中用的最多的是：stylus 和 CSS 的混合。为啥呢？第一：感觉项目只使用一种样式语言可能比较少，每种样式语言都是很大的缺陷，所以会使用 Stylus 来弥补 CSS 上的不足。第二：Stylus 是 node.js 社区推出来的，可能在打包构建等方面更有优势吧。最后退一步说，其实 CSS 的预处理语言的语法大多数情况下都是相同的，所以你需要用哪个，都可以！

- 1、使用 scope 关键字，约束样式生效的范围。[参考这里](#)
- 2、引入 .styl 文件：使用 `@import '../xx/xxx.styl'` 相对路径的形式。逗号结尾，并且在最后一个引入，加一个换行。
- 3、避免使用标签选择器。因为这个在 Vue 中，特别是在局部组件，效率特别低，损耗性能，建议需要的情况，直接定义 class。[参考这里](#)
- 4、非特殊情况下，禁止使用 ID 选择器定义样式。有 JS 逻辑的情况除外。
- 5、CSS 属性书写顺序：先决定定位宽高显示大小，再做局部细节修饰！推荐顺序：定位属性(或显示属性，`display`)>宽高属性>边距属性(`margin, padding`)>字体，背景，颜色等等修饰属性的顺序定义。

这样定义为了更好的可读性，让别人只要看一眼就能在脑海中浮现最终显示的效果。一下给出常用的定义示例：

```
.class-name {  
  position: fixed;  
  top: 100px;  
  left: 0;  
  right: 0;  
  bottom: 0;  
  display: block;  
  width: 100%;  
  height: 100%;  
  margin: 10px;  
  padding: 10px;  
  font-size: 14px;  
  color: #000;  
  background-color: red;  
  border-radius: 2px;  
  line-height: 1.42;  
}
```

# 注释规范

## 什么东西应该注释？

写注释固然很重要，但是很多时候我们真的需要注释吗？好好想一下这个问题。以下图片截取自：[什么时候应该避免写代码注释？](#)

那么，为什么这样的注释反而不好呢？简而言之是因为，我们会因为有注释而放任编写坏的代码！正如你所看到的，注释有时候反而激励了我们去写一些不整洁的代码。

另一个原因是注释会误导我们。有多少次你已经改变了代码，却忘了改旁边的注释？别否认，这种事时常发生。这就是为什么你经常听到这样一句话，“真理只存在于代码中”。

那么，什么时候你不应该写注释呢？

有一个经验法则就是，无论何时你发现自己要使用注释来解释这段代码是用来干什么的时候，那么基本上就是你的代码需要重构以变得更整洁的时候。

典型的解决方案

现在你知道为什么有时候反而要避免写注释了，那么有什么解决办法吗。事实上，目前还没有一个有效的解决方案，但是你可以清洁你的代码，这样你（以及其他）就可以在没有注释的情况下也能理解它了。

为了用可读的代码替换掉注释，通常我们的典型解决方法是使用提取方法重构（Extract Method refactoring）。这种重构方式是我最喜欢的。对此我也写过一篇博客，里面有完整的例子：《Break Your Method Into Smaller Ones》。

所以很多时候我们写的都是一些没有用的注释，不但没有用，反而还会误导人！我觉得可以写成注释的大概有这么几样：

- 1、公共的函数方法、常量和重要变量的定义。
- 2、业务系统中，晦涩难懂的特殊需求。
- 3、多步骤的流程。最好用1, 2, 3, 4… 标明顺序。
- 4、复杂的算法。

其他没有用的东西直接删除，保持代码的整洁性。不要用注释的办法来设法回退，那是 git 版本管理等工具应该完成的事情。

## 注释的格式

### HTML注释

在 html 中主要使用的注释方式就只有一个： `<!-- write your comment! -->`

注意在注释的前后各有一个空格。

### CSS注释

在 CSS 中，注释的方式主要有：`/\* write your CSS comment! \*/`

注意在注释的前后各有一个空格。

## Javascript

在 JS 中，因为注释的东西，内容比较多，按照前面列出的功能，分别说明。

### 行级注释

行级注释就是指的双正斜线后面的内容，双线后面需呀放置一个空格。

```
// 正确的注释
//错误的注释，双斜线后面没有空格
```

### 变量声明注释

1、如果是在类似 Vue 项目的 data 属性中的变量，直接用行级样式跟在后面即可。

```
data () {
  return {
    rightExample: 'yes', // 注释直接写这里
    // 我是 errorExample 变量的注释，错误形式
    errorExample: 'no'
  }
}
```

2、如果是在类，构造函数，或者常量定义中的变量，使用块级注释。

```
/**
 * 错误码常亮定义
 * @type {number}
 */
const ERR_CODE = 0
```

### 函数声明注释

不必要在每一个函数都写注释，但是在公共函数，还是建议补全注释，让后面的人不需要重复早轮子。

函数头注释中必须写以下内容：

- 函数的功能
- 函数的入口参数类型
- 函数的返回值类型
- 函数的算法或者业务逻辑

```
/**
```

```
* 获取 DOM data 属性的值
* @param {DOM} el dom对象
* @param {String} name 需要获取属性的名字
* @param {String, Number} val 可选, 存在是表示设置属性的值
* @returns {String}
*/
function getData (el, name, val) {
  const prefix = 'data-'
  let prop = prefix + name
  if (val) {
    el.setAttribute(prop, val)
  } else {
    return el.getAttribute(prop)
  }
}
```

参考资料:

<https://jingyan.baidu.com/article/3ea51489abc0f152e61bba2e.html>

<https://www.jianshu.com/p/1a0cf697b9bb>

# Git 提交代码约定

提交格式：[关键字]:空格[内容]

解释：操作关键字+英文冒号+一个空格+操作内容

以下是可以使用的操作关键字：

**created**: 创建文件夹， JS 文件， 等等

**deleted**: 删 除文件夹， JS 文件， 等等

**modified**: 修改内容或文件

**add**: 添加文件内容， 和新建不同

**fixed**: 修复BUG， 缺陷等

小步快跑， 尽量完成一个功能， 提交一次代码！

# 数据中心说明

**注意：**使用数据状态管理意味着你修改数据必须使用：Mutations 或 Actions，读取数据必须通过：Getters，不能在数据中心外面调用私有的方法修改数据，这会使你修改数据的路径变长。所以需要你权衡出数据由中心带来的不便和你要解决问题的规模，复杂程度之间的关系，然后在决定用不用。最后，如果你决定使用Vuex，我假设你已经试过了其他的方式，并且效果不理想、愿意接受 Vuex 带来的不便，那么，欢迎！

这个是封装的数据中心的文件结构。别担心，现在不懂没关系，我们会在后面，通过一个小小的购物车的例子，一步步带领你熟悉 Vuex 的使用，并说清楚每个文件的作用。

|                     |                     |
|---------------------|---------------------|
| └ store             | Store 目录            |
| └ index.js          | 运行本地构建服务器，可以访问构后的页面 |
| └ state.js          | 数据仓库，数据中心的数据源       |
| └ mutation.types.js | 定义 Mutations 的函数的名字 |
| └ mutations.js      | 定义 同步 操作数据源的函数      |
| └ actions.js        | 定义 异步 操作数据源的函数      |
| └ getters.js        | 定义从 State 中获取数据的函数  |

最后，如果你还对数据中心的一些基本概念，比如：数据中心是什么，如何工作的，如何提交一个 Mutations，Actions，…，还不是熟悉，请先看 [Vuex官网](#) 其实很多东西在官网讲的可能会更详细，这个文档是建立在你有一定认知的基础上，能够帮助到你快速进入到生产开发的状态！

以后可能会频繁使用 ES6 的语法，如果你还不是很熟悉，可以移步阮一峰老师的：[ECMAScript 6 入门](#)

# 第一步：数据中心（State）

首先你应该在脑海里有一个认知，数据中心（Vuex）是什么？

按照官网的说法：Vuex 使用单一状态树——是的，用一个对象就包含了全部的应用层级状态。

简单理解就是：它一个全局的对象。只是这个对象有点特殊，挂载了很多的变量，你不能直接修改，必须通过特定的方法读取/修改它们。那么，既然这个变量是全局的，这就意味着，一个应用里面只会拥有一个数据中心的实例。我们按照习惯，用 store 表示。

好了，有了这个认识，我们回到主题，讲解 Vuex 的数据源：State.

首先我们应该创建一个数据源（state），它是数据中心的核心部分，存放数据的地方。初始化很简单，就像这样：

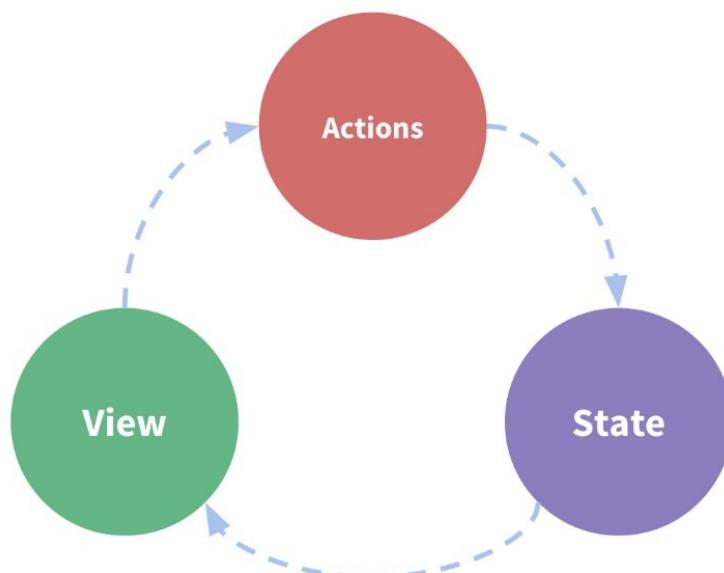
```
// state.js
const State = {}
export default State
```

我们声明了一对象，并把它暴露出来。目的是为了让数据源独立出来，以便日后更好的维护，和模块化。

这里有一个比较容易混淆的点，就是：数据中心（store）和数据源（state）的关系。特别是在后面传参数的时候，什么时候传递的是 store，什么时候是 state 一定要做好区分：

**store** 是数据中心的实例，能够访问到所有数据中心的数据；

**state** 只是数据源，是 store 中的一个属性，牢记它们区别。



我们来一步步实现前面说的购物车。我们可以在 State 里面添加一个变量，保存购物车的数据，就像这样：

```
// state.js
const State = {
  cardData: [] // 保存购物车的数据
}
export default State
```

好了，到这里，关于 state 的东西就这么多，比较简单。还是那句话，如果你还是不懂，可以先去参考官网的：[Vuex](#) 部分。

我们有了数据源，接下来我们看看：[如何向 State 中添加数据](#)。

# 同步修改数据源的数据 (Mutations)

经过上面一节，你已经迈出了第一步：如何定一个数据源。仓库是有了，但是我们如何在里面存储东西呢？

这就是我们这节的主要内容：同步修改 State 中的数据。

修改 State 里面的数据有同步方法和异步方法，不能混用，在 Vuex 中有严格的规定。稍后将会说异步方法。

根据 Vuex 的规定，修改 State 的数据只能是通过一个 Mutations 或是 Actions。所以我们通过函数的方式是最合适的。就比如：

```
new Vuex.Store({
  state: {
    cardData: []
  },
  mutations: {
    addSomething (state, arr) {
      // 变更状态
      state.cardData = arr
    }
  }
})
```

这样，写在 mutations 对象里面，Vuex 会把 State 数据源作为第一个参数传递到函数里面，后面的参数作为载荷提交，其实就是你给函数传递参数。

我们这样写以后，在自己的组件中，通过 Vuex 提供的语法糖：

```
import { mapMutations } from 'vuex'

// your component
methods: {
  ...mapMutations([
    'addSomething'
  ]),
  yourMethods () {
    this.addSomething([
      {
        name: 'test'
        // ...
      }
    ])
  }
}
```

使用即可，感觉特别方便。

优化点：

这里根据官网的推荐，当多人协作时，可能 Mutations 里面的方法会很多，并且很不好维护，所以，推荐另外使用一个文件来专门定义 Mutations 的行为。这个就是目录中：mutation.types.js 的由来。里面防止的全是一些方法名字的常量。引入该文件后，修改可能是这样的：

```
// mutations.types.js
/**
 * 设置购物车数据的 Mutations
 * @type {string}
 */
export const SET_CARD_DATA = 'SET_CARD_DATA'
```

```
// mutations.js
import * as types from './mutations.types.js'

// ...
mutations: {
  [types.SET_CARD_DATA] (state, arr) {
    // 变更状态
    state.cardData = arr
  }
}
```

调用方式：

```
import { mapMutations } from 'vuex'

// your component
methods: {
  ...mapMutations({
    addSomething: 'SET_CARD_DATA' // 创建一个函数的映射
  }),
  yourMethods () {
    this.addSomething([
      {
        name: 'test'
        // ...
      }
    ])
  }
}
```

注意：

- 1、mutations 中的数据是响应式的，所有对于数组对象的赋值需要注意；
- 2、再次提醒，Mutations 里面不能有异步的操作，比如请求一个接口获取数据，读取本地文件的异步行为！



# Mutation.types

这里主要说说，为啥需要单独一个文件用来定义常量。感觉官网的讲的非常详细，话不多说，直接上图。如果你要看原文，[点击这里](#)

## # 使用常量替代 Mutation 事件类型

使用常量替代 mutation 事件类型在各种 Flux 实现中是很常见的模式。这样可以使 linter 之类的工具发挥作用，同时把这些常量放在单独的文件中可以让你的代码合作者对整个 app 包含的 mutation 一目了然：

```
// mutation-types.js  
export const SOME_MUTATION = 'SOME_MUTATION'
```

```
// store.js  
import Vuex from 'vuex'  
import { SOME_MUTATION } from './mutation-types'  
  
const store = new Vuex.Store({  
  state: { ... },  
  mutations: {  
    // 我们可以使用 ES2015 风格的计算属性命名功能来使用一个常量作为函数名  
    [SOME_MUTATION] (state) {  
      // mutate state  
    }  
  }  
})
```

用不用常量取决于你——在需要多人协作的大型项目中，这会很有帮助。但如果你不喜欢，你完全可以不这样做。

# Actions

说完了同步的修改数据方式，我们来说说异步的修改方式。在实际的项目开发中，需要的情况也不总是同步的，很多时候是异步的。比如：我们需要想接口请求数据，或者我们需要做数据的离线缓存。等等，这里就需要我们使用一个异步的场景，那就是：Actions。

Actions 和 Mutation 基本用法一样，但是有些区别：

1、Action 提交的是 mutation，而不是直接变更状态。

2、Action 可以包含任意异步操作。

这两点有点区别。还记得前面的 mutations 同步提交的时候只支持一个传入的 state 参数吗？在 Actions 里面传递的参数不同了：

```
actions: {
  saveCardData ({commit, store}, paloyloadObj) {
    console.log(store) // 传入数据中心的对象，你可以拿当前数据中心的实例
    axios.post('url', paloyloadObj).then((res) => {
      if (res.status === 200) {
        commit(paloyloadObj) // 保存成功，将数据同步到数据中心
      }
    })
  }
}
```

这里一大区别就是，传递的参数不一样了，Actions 提交的是一个向数据中心请求修改数据的操作，不是直接和 Mutations 一样，直接去改数据 这里可以感受一下同步和异步的差别。

调用的方式还是和Mutation一样，Vuex 给我们封装了一个语法糖：mapActions 拿过来就是可以直接用，非常简单！

# 读取数据中心数据 (Getters)

前面说了，如何定义数据中心，如何修改数据中心中的数据，同步，异步。现在我们来说说：何如读取数据中心的数据。

这个很简单，根据官网的使用方式，主要有：定义为一个计算属性；通过Vuex 提供的语法糖。这里我们只说用的最多的：mapGetters.

首先我们需要在 getters 中定义一个获取数据的函数：

```
getters: {
  getCardData (state) {
    return state.cardData
  }
}
```

然后在组件中：

```
import { mapGetters } from 'vuex'

methods: {
  ...mapGetters[
    'getCardData'
  ]
}
```

调用一下方法就可以获得数据，很简单。

但是我们通过把，getter 分离出来，加上ES6 的语法，这里的函数定义会更简单：

```
// getters.js
export const getCardData = state => state.cardData
```

调用方式还是一样，但是更加简洁明了。

我们这里讲了这么多，希望那你不不要混淆，感觉每个环境都没有那么重要，所以下一节，你将会看到一个完整的例子，到时候你会体会到，这样定义的初衷。

# Index

前面一直在讲Vuex 的每个模块的作用，可能听着有点犯迷糊，不知道是用来干嘛的，所以这里做一个汇总。

一个完整数据中心的结构应该是这样的：

```
export default new Vuex.Store({
  state, // 数据源，对应这State的部分
  mutations, // 同步修改数据
  actions, // 异步修改数据
  getters // 获取数据
})
```

这里面不一定都需要有，其中的actions 是可以省略的，如果没有涉及异步的情况。

所以结合前面的知识，当我们拆封 Vuex 的时候，我们大概会看到一开始，首页中的目录结构，并且这个index.js 最为Vuex 的入口，我们完整的看到应该是这样的：

```
import Vue from 'vue'
import Vuex from 'vuex'
import state from './state' // 引入数据中心，其实就是一个对象
import * as actions from './actions' // ES6 语法，引入全部的 Actions，并且命名为：actions
import * as getters from './getters' // ES6 语法，引入全部的 Getters，并且命名为：getters
import mutations from './mutations' // 引入全部的 通过常量定义的 mutations
import createLogger from 'vuex/dist/logger' // 引入开发时的控制台数据跟踪日志工具

// 用于开发测试，上线时关闭
const isDebug = process.env.NODE_ENV !== 'production'

Vue.use(Vuex) // 挂在到Vue 的原型链上

export default new Vuex.Store({
  state,
  actions,
  getters,
  mutations,
  strict: isDebug,
  plugins: isDebug ? [createLogger()] : []
})
```

好了，所有关于数据中心的定义就讲到这里。感觉这里的很多东西，只能够帮你解决项目上遇到的部分问题，帮你快速了解项目。如果你需要一个详细可靠的文档，还是建议你参考官方文档。

# 路由定义

这里放置项目路由的定义

# 这里放置整个项目的规范

各个模块的规范

# 项目前期准备

放置项目开始前期做的初始化工作。

# 首页

已实现：

- 1、页面主体内容自动滚动，只要将内容渲染在 scroll 组件中即可；
- 2、上拉刷新实现，在回调：pullingDownGetNewData 函数中处理就好；

注：需要把请求成功的数据回填 scroll 组件（填到 data 属性中），从而触发页面重新计算高度

- 3、下拉加载，处理方式和上拉刷新一样，注意请求结果为空时，手动调用刷新组件函数即可；